Diagnosing Cyber-Physical Systems with CatIO

Edi Muškardin and **Ingo Pill** and **Franz Wotawa**
Christian Doppler Laboratory for Quality Assurance Methodologies for Cyber-Physical Systems
Institute for Software Technology, Graz University of Technology
Inffeldgasse 16b/II, 8010 Graz, Austria
e-mail: {edi.muskardin, ipill, wotawa}@ist.tugraz.at

Abstract

Diagnosing cyber-physical systems is intrinsically challenging due to the complex interaction between physical and cyber components. In this manuscript, we will demonstrate the capabilities of our CatIO framework supporting a designer in diagnosing such systems and assessing the quality of the process. Via a specialized interface, the diagnosis engines can be fed with observations from Modelica simulations, where Modelica is a perfect means for modeling CPSs. Via these simulations we can assess the diagnostic reasoning's quality even early on during development, and we can automatically create a diagnostic model when simulating injected faults and their effects. CatIO also allows designers to interact with a simulation at run-time, such as to asses repair and compensating actions derived by a CPS's control logic.

1 Introduction

Intelligent cyber-physical systems (CPSs) are nowadays deeply integrated in our everydays life. With future developments of artificial intelligence and the Internet of Things, our reliance on their ability to autonomously discharge their duties is likely to continue to grow even further. The interconnections between a cyber system and its environment makes a CPS quite complex though and introduce several challenges in areas like testing, verification and validation, and model-based diagnosis [13]. That is, those systems have to be aware of their current state, that of the environment, and they have to know how to deal with occurring problems.

Due to their features, intelligent CPSs can be found in many application domains, including aerospace engineering, automotive industry, healthcare, manufacturing and many more. Depending on the application, human control over a CPS is often limited to the roles of observer and task definition. Once their tasks have been defined, we often expect those systems to be fully autonomous, safe, secure and fault-tolerant/fail-operational, i.e., dependable. That is, if the compensating or repair actions for mitigating experienced issues can be performed autonomously, we can save time, resources and money. Obviously, this requires us to heavily test a CPS and its control logic, such as to establish the required trust. We also need a diagnostic reasoning engine that can provide the CPS with the diagnostic data needed to make smart decisions for mitigating the undesired effects.

Our CatIO[1] framework was designed for exactly that purpose. The architecture as outlined in [8] connects consistency-oriented model-based diagnosis and abductive diagnosis with Modelica [6] models, such as to support diagnosing a CPS using Modelica simulations during a CPS's development—of course one can use also "real" observations from the final CPS. We also support the automated generation of abductive diagnosis models via simulation and fault injection [4], and there is also a dedicated interface to connect a CPS's control logic to a simulation. Via the latter, the control logic can draw on the simulation and diagnosis results for deriving the desired compensating actions and then implement them by feeding them into the simulation.

2 Preliminaries

The ratio behind model-based reasoning is to reason about a system's behavior using a system's model. Model-based diagnosis (MBD) [11; 3] as such a technique allows us to reason about faults present in the system by considering a specific model and observations about its actual behavior. This dedicated model comprises a set of components and also interconnections between them. Each component can be seen as a subsystem of the system as a whole, with its in- and outputs connected to other components or being a system's primary in- and outputs. Each component c features a health state $h_c \in H$—a proposition that states whether the component is OK and thus behaves as expected ($h_c = \top$ means c is OK). Consequently, health states can be seen as assumptions about the correctness of the components/system and, if the system does not behave as expected, they are used to isolate (a) root cause(s).

When using the classic *weak* fault model, we describe only the components' correct behavior. That is, if a component is faulty ($h_c = \bot$), there is no assumption on this component's behavior. Consequently we do not require any knowledge about possible faults and their consequences. Whenever we include also such knowledge, we implement *strong* fault models [4].

Once a diagnostic model has been derived, we can use it to tackle the diagnosis problem of finding explanations for unexpected/faulty behavior. In particular, once we find that a system's actual behavior contradicts the expected behavior described in the diagnosis model—when assuming that all components are OK. Intuitively, a contradiction entails

[1] From "**Ca**usarum Cogni**tio**" which is Latin for "(seek) knowledge of causes"

that some component(s) must behave abnormally (or that the model is incorrect).

Definition 1. *A diagnosis problem can be described as a tuple (SD, H, OBS), where the tuple (SD, H) describes the system to be diagnosed and OBS is a set of observations concerning its behavior.*

A solution Δ to a diagnosis problem is a set of components c_i which, when assumed that these c_i behave abnormally (s.t. $h_{c_i} \in \Delta$), explains the conflicts between expected and observed behavior [3; 11].

Definition 2. *$\Delta \subseteq H$ is a diagnosis for a diagnosis problem (SD, H, OBS) if and only if $SD \cup OBS \cup \{h_i | h_i \in H \setminus \Delta\}$ is consistent (satisfiable) and there exists no $\Delta\prime \subset \Delta$ that is also a diagnosis.*

In consistency-based MBD, we require the observations to be consistent with our expectations as stated in Def. 2 In this context, we have that a set of logical statements is consistent iff all statements can be true at the same time. Therefore we usually describe components, their interconnections and behavior in some logic format. If the system behaves as expected, the union of all logical sentences in *SD* and *OBS* will be consistent with the assumption $\Delta = \emptyset$. An inconsistency entails that some sentences in the model cannot be evaluated as truths. A conflict is then a set of logical sentences that are inconsistent, where for MBD we focus on conflicts in the assumptions that components c_i are OK.

Definition 3. *A conflict C for a diagnosis problem (SD, H, OBS) is a subset of H such that $SD \cup OBS \cup \{h_i | h_i \in C\}$ is inconsistent. If no proper subset C' of C is a conflict, then C is a subset-minimal conflict.*

We can compute diagnoses for some diagnosis problem as the minimal hitting sets of the set of conflicts, where it suffices to focus on the subset-minimal conflicts [11; 3]

Definition 4. *A hitting set Δ for a set CS of conflicts as of Def. 3 is a subset of H such that $\Delta \cap C_i \neq \emptyset$ for all $C_i \in CS$. Δ is a minimal hitting set if and only if no proper subset of Δ is a hitting set as well.*

Taking conflicts between the observed behavior and the expected one (as described in the system description) into account, we can easily derive the diagnoses using algorithms like RC-Tree [9]. Many algorithms, including RC-Tree, can compute the conflicts on-the-fly by calling a theorem prover (like a SAT solver) to check the consistency of the observations with the system model for some diagnosis hypothesis (some $\Delta \subseteq H$) as described above, and ask the solver for a conflict set (a minimal unsatisfiable core in H) if this is not the case.

CatIO supports abductive diagnosis problems in the form of Propositional Horn Clause Abduction Problems (*PHCAPs*). Each *PHCAP* formulates our knowledge about a system in a knowledge base (KB). For a *PHCAP*, we express this knowledge base as a set of Horn clauses over propositional variables *PROPS*, where a Horn clause is a disjunction of literals such that at most one literal is positive. Propositional variables abstractly denote some value in the system or a particular mode of a component. In context of this paper, a hypothesis corresponds directly to causes (of experienced issues) and we thus use the terms interchangeably. Formally, we can define a knowledge base and a *PHCAP* as follows [5]:

Definition 5. *A knowledge base (KB) is a tuple (A, Hyp, Th) where A $\subseteq$ PROPS denotes a set of propositional variables, Hyp $\subseteq$ A a set of hypotheses and Th a set of horn clause sentences over A.*

Definition 6. *Given a knowledge base (A,Hyp,Th) and a set of observations Obs $\subseteq$ A, the tuple (A, Hyp, Th, Obs) forms a propositional horn clause abduction problem (PHCAP).*

A solution to a *PHCAP* is a set of hypotheses that allows deriving the given observations [5]:

Definition 7. *Given a PHCAP (A, Hyp, Th, Obs), a set Δ $\subseteq$ Hyp is a solution if and only if $\Delta \cup Th \models Obs$ and $\Delta \cup Th \not\models \bot$. A solution Δ is parsimonious, in other words minimal, if and only if no set $\Delta\prime \subset \Delta$ is a solution.*

A solution Δ to a *PHCAP explains* the given observations Obs such that we refer to it also as abductive diagnosis.

While diagnosis models are used to reason about a system's correctness (usually at some abstract level), simulation models describe a system in great detail. For the latter a designer would use detailed equations that describe the intrinsics of system's components and their interaction. Digital circuits can certainly be expressed using Boolean logic if we are interested in their functionality, but she will use differential equations if we're interested in the exact voltages. The latter is certainly true for the physical components of a CPS and its environment, where we aim to precisely investigate their continuous behavior over time.

Definition 8. *A simulation model $\mathbf{M} = (COMP, MODES, \mu, I, O, \mathbf{P})$ is a tuple that comprises a finite set of components COMP, a finite set of modes MODES that contains at least the correct mode ok as element, a function μ : COMP $\mapsto$ MODES mapping components to their featured modes, a set I of variables considered as inputs, a set O of variables considered as outputs, and a Modelica model $\mathbf{P}$ that allows us to set an individual mode $m \in \mu(c)$ for each $c \in$ COMP.*

When we inject one or more faults into a system [14], we have to describe a fault model for the corresponding component(s), in order to simulate the consequences. Fault injection can thus be formulated as a (fault) mode assignment, and a simulation can be seen as a function that computes all of P's variables' values over time—with respect to the system inputs, mode assignment and simulation run time.

Definition 9. *Let TIME be a finite set of points in time. A mode assignment Δ is a set of functions δ_i : COMP $\times$ TIME $\mapsto$ MODES that assign to each component $c \in$ COMP for each time point $t \in$ TIME a mode $m \in \mu(c)$, i.e., $\Delta = \{\delta_1, \ldots, \delta_{|TIME|}\}$ where $\forall i \in \{1, \ldots, |TIME|\}$: $\forall t \in TIME : \forall c \in COMP : \delta_i(c, t) \in \mu(c)$.*

3 About CatIO

CatIO[2] was developed out of the need for a unified framework that eases the development of diagnostic models and allows a smooth interconnection of a diagnostic process and simulation scenarios. The development of a diagnostic model is a cumbersome task in itself, and lacking a unified way of testing said model against a simulation of a fault injected into the system, MBD remains underrepresented in the industry tool base. With CatIO, we offer such a solution

[2] https://github.com/EdiMuskardin/CatIO-MBD-Framework

and aim to demonstrate how a designer can conveniently develop a diagnosis system and test/assess its capabilities.

CatIO supports two modeling paradigms for developing diagnostic models. For developing consistency-based MBD models, we support descriptions in propositional logic, whereas for abductive models we support using a set of Horn clauses (as supported in PHCAPs) in a PROLOG-like syntax. All diagnostic models can, of course, be manually tested by entering observations. By obtaining the observations from some Modelica simulation scenarios investigated with CatIO, a designer can quantitatively assess the diagnostic capabilities of a model early on during its development.

The diagnostic features of CatIO are extended by enabling the consideration of compensating actions against in a simulation at run-time. The simultaneous development of the model and the compensating actions featured by an autonomous CPS certainly fosters a faster development of both. The execution of compensating and repair actions depends on the information obtained from diagnostic reasoning. Considering this information can foster development of the diagnostic model which is able to provide sufficient information about the faults.

Exploiting a fault injection and simulation concept as outlined in [14], allows users of CatIO to automatically generate abductive models. Such automatically generated models can be investigated in Modelica simulations of the CPS in order to assess their diagnostic capabilities. The supported manual extension by physical impossibilities or additional behaviors of these models can lead to improved diagnostic capabilities.

3.1 Using the CatIO framework

To use CatIO capabilities to their intended and full extend, we assume that designer has a Functional Mock-Up Interface (FMI)[2] defining the model. Model can be accompanied by test-bench which defines all inputs over time or it can be "input-oriented", so that all inputs to the model are defined with CatIO. CatIO distinguish between two different Modelica model/simulation paradigms. First, a Modelica model of a system is developed together with a corresponding test bench, which assigns values to the simulation model over time. Second, an "input-oriented" model is a Modelica model where all changeable variables are set as inputs using external tools like CatIO. Such variables are system inputs, system parameters and mode assignment variables. Both paradigms are fully compatible with CatIO, where the latter is preferred as it enables a manual or programmatic creation of simulation scenarios via assigning inputs, parameters or modes over time. Using the latter variant, FMI can be used as a digital twin of a cyber-physical system.

In order to use CatIO for diagnosis, raw data of the simulation has to be mapped to the propositions found in the model, and more precisely to observations. Note that simulation data can be of any type and value, whereas diagnostic models consist of a finite set of propositions, which are encoded as strings. Propositions can be represented qualitatively, e.g., as deviation model or a value abstraction. To map raw simulation data to proposition, a designer has to implement a `Encoder` interface in CatIO. Propositions obtained from the `Encoder` are observations of the system.

With the diagnostic model and a corresponding `Encoder` a designer may couple diagnostic capabilities with Modelica simulation directly. In case of observed faults and corresponding diagnosis, designer may further be interested in exploring available consequences of compensating and repair actions. For this purpose, a `Controller` interface enables run-time interaction with the simulation. It defines a set of actions performed on the simulation over time. Length of compensating and repair action is user defined and may change over time depending on the systems response.

Besides diagnosis and repair functionality CatIO offers easy model construction using the `Diff` interface with a corresponding `diff` method. This method is used in CatIO to compare a fault free and a fault injected simulation. Once a discrepancies between simulations are detected, they are mapped to propositions that are used to automatically construct an abductive model in form of $faulty_1(comp)...faulty_n(comp) \rightarrow discrepancy$. Combination exploration of possible simulation inputs, parameters and fault state is used to define simulation scenarios which are used to generate abductive model. Depending on designer's preferences, he or she can compare propositions obtained from encoder or raw simulation values in `diff`. The former can be used to generate deviation models, while latter enables the generation of qualitative models where abstract value representations are used.

4 Running Example

To illustrate how CatIO can be used, we make use of a robot's differential drive (cf. Fig. 1). Such a drive is characterized by having two individually controllable wheels. Intuitively, if both wheels spin with the same speed, the robot will move in a straight line. If one wheel spins faster than the other, it will turn into the direction of the slower spinning one instead.

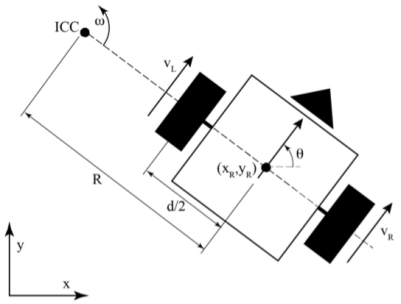

Figure 1: Schematic of differential drive robot

Let us now describe the kinematics in more detail. To this end, let us introduce a coordinate system (X_R, Y_R), where the robot's heading is described via the angle θ to the X-axis. The parameter d defines the distance between the wheels, and the desired wheel spin is controlled via "input" voltages V_R and V_L respectively. A robot's rotation from the Instantaneous Center of Curvature (ICC) at radius R (to the robot's center) is described via the rotational speed ω.

If the drive works as expected, the two wheels spin with the speeds defined by the input voltages V_R and V_L. The model contains also "output" voltages capturing the actual spinning speeds, so that for a correctly working drive, the respective voltages will be identical. If, however, a wheel spins *faster* or *slower* than expected (due to some problem), the voltages will differ and we will be able to observe a compromised heading (cf. Fig. 2).

5 Demonstrating CatIO's Capabilities

5.1 Data Selection and Scenario Generation

Using CatIO, a designer can run simulations via an FMI. As outlined in Sec. 3.1, we need to map the signals and their values in a simulation to the format suitable for the diagnostic model. CatIO's graphical user interface (GUI) supports the designer in performing this step. There a designer can select a set of those variables found in the FMI that should be read for every simulation and fed into the `Encoder` described in Sec. 3.1. These "Read" will then define the inputs/probing points/sensory data as observations for diagnosis of the CPS. For our running example, we observe the input and output voltages coming from FMI of the mobile robot.

In terms of input scenarios for simulation a designer can either define them manually or use an automated generation procedure. The CatIO GUI allows a designer to automatically generate simulation scenarios using a combinatorial exploration of all system inputs, parameters and health state options like used in combinatorial testing [7] (using the ACTS[15] tool library). The designer selects those variables/signals that shall be assigned automatically and then he or she has to define a finite set of representative values for each of those variable. Constraints over variables values, n-wise testing and minimal number of correct components can be defined to further reduce the size of automatically generated scenario suite.

For a manual specification of a scenario, a designer can populate a dynamically generated table (depending on the length of the simulation and number of inputs) analogue to Table 1 via the GUI. All input values of a simulation have to be initialized and whenever a value change is desired from some time step t, the new value is entered for t (and stays unchanged until another change occurs).

Table 1: Manual specification of simulation scenario.

Step	d	mode(Lw)	mode(Rw)	input(Lw)	input(Rw)
0	2	ok	ok	3	3
5			faster		
15			ok		

Table 1 depicts a simulation scenario where we injected a fault at time step 5 of the simulation as well as correction step at time step 15. Input values for both wheels ($input(Rw), input(Lw)$) and distance between wheels (d) remain unchanged through simulation.

5.2 Modeling with CatIO

For developing the simplest model of the differential drive robot shown in Section 4, we make use of the following underlying thoughts. If the differential drive robot is behaving abnormally, its diagnostic model has to be inconsistent with observations. Observation abstraction and modeling is the starting point of the development of a diagnosis model. Based just on two input voltages that are the inputs of the our robot drive system, we can model observations and models with different levels of abstraction.

In CatIO the following grammar is used for developing a consistency based model where models are expressed using propositional logic.

Listing 1: Grammar for consistency based models

```
Health State ::= [A-Z][A-Za-z0-9_]*
Variable ::= [a-z0-9_@][A-Za-z0-9_]*
True ::= '$true'
False ::= '$false'
Negation ::= '!'
Operator ::= '&' | '|' | '->' | '<->'
Literal ::= Variable | HealthState | True | False | Negation
    Literal
Formula ::= Literal | '(' Formula ')' | Formula Operator
    Formula
Clause ::= Literal (',' Literal)* '.'
ConsistencyModel ::= (Formula '.')+ | (Clause '.')+
```

As stated in Section 4, the normal behavior of the wheel is characterized by equal input and output value. We encode such behavior using the propositions $equalInputOutput\_Left$ and $equalInputOutput\_Right$ for left and right wheel, respectively. If the wheel is behaving abnormally the propositions are negated. Because these propositions are directly related to the "weak" fault-mode of the wheel, the diagnostic model can be expressed as given in Listing 2. Capitalized propositions are the result of the diagnostic procedure because they represent health state variables. If at any point an observation stating a proposition describing the relation between the wheel input and output evaluates to false, then the corresponding wheel associated with that proposition is a single diagnosis.

Listing 2: Weak fault model

```
!AbLeftWheel -> equalInputOutput_Left.
!AbRightWheel -> equalInputOutput_Right.
```

Note that different modeling of observations results in a diagnostic model with different diagnostic capabilities. When comparing inputs and outputs of both wheels, we can determine the wanted and actual direction of the differential drive robot. If the wanted and actual direction are not equal, a fault has occurred and diagnostic model shown in Listing 3 is used to compute diagnoses.

Listing 3: Strong fault model

```
wantedStraight & actualLeft -> SlowerLeft | FasterRight.
wantedStraight & actualRight -> FasterLeft | SlowerRight.
wantedLeft & actualStraight -> FasterLeft | SlowerRight.
wantedLeft & actualRight -> FasterLeft | SlowerRight.
wantedRight & actualLeft -> SlowerLeft | FasterRight.
wantedRight & actualStraight -> FasterLeft | SlowerRight.

// physical imposibilities
wantedStraight & wantedLeft -> $false.
wantedStraight & wantedRight -> $false.
wantedRight & wantedLeft -> $false.
actualStraight & actualLeft -> $false.
actualStraight & actualRight -> $false.
actualRight & actualLeft -> $false.
```

5.3 Diagnosing with CatIO

As already noted CatIO obtains observations during simulation for diagnosis. As stated in Section 3.1, CatIO takes the raw simulation data and map them to the corresponding propositions. For this purpose the values obtained from simulation are passed after each time step to the `Encoder`, which itself performs the mapping.

Consistency-based model makes use of propositional logic, which is transformed to an equivalent conjunctive normal form (CNF) in CatIO. This CNF model is itself mapped to its corresponding representation in DIMACS standard format where each propositional variable of the CNF model is mapped to an integer. Model encoded in DIMCAS standard is used afterwards in combination with PicoMUS, a minimal unsatisfiable core extractor based on PicoSAT[1] SAT solver for computing conflicts. In

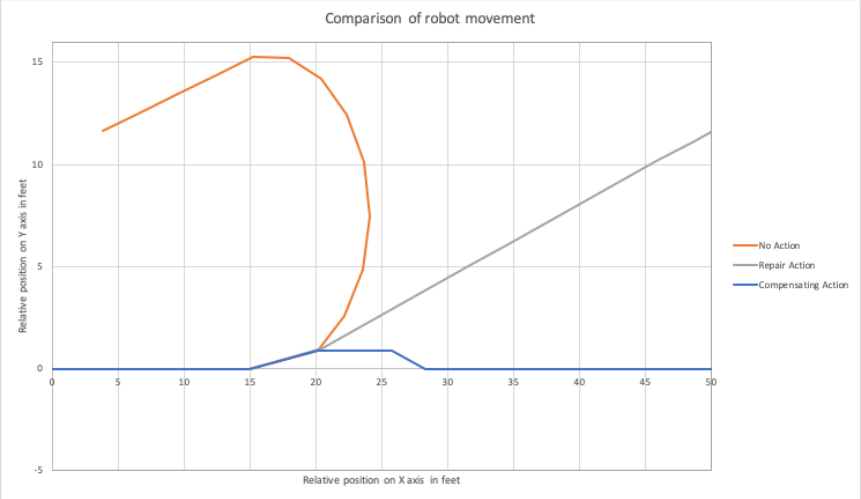

Figure 2: Comparison of robot's movements.

CatIO the consistency-based diagnosis procedure can be performed making use of three separate configurations.

First, the diagnosis procedure can be performed after each time step of simulation. Observations obtained for each step are added to the CNF. If they are unsatisfiable together with the model, consistency-based diagnosis is computed with RC-Tree algorithm. Note that abductive diagnosis can only be performed in this configuration.

Second mode of diagnosis assumes that faults in the simulation are persistent. Persistent diagnosis returns all possible diagnosis with the assumption that the occurrence and duration of each fault is irrelevant and faults persist over time. The persistent fault-mode diagnosis procedure is invoked at the end of the simulation. After each time step the diagnostic model is expanded by adding all clauses that are indexed with timing information to make them unique except health state assumptions. Diagnosis after simulation returns all faulty components that explain all discrepancies in such expanded model.

Finally, consistency-based diagnosis can be performed with the assumption that the faults are intermittent. Intermittent faults do not necessarily persist over time as the health state of the component may revert to the correct one. Intermittent consistency-based diagnosis considers the duration and moment of occurrence of each fault. Intermittent diagnosis is more precise than persistent-faults diagnosis and diagnosis performed after every time step of the simulation as it returns the traces explaining all discrepancies. Trace is a list of fault modes with corresponding time step. The intermittent fault model is constructed by incrementing and adding all integers representing propositions and observations after each time step of simulation. Run-time of intermittent diagnosis procedure increases exponentially as the number of possible faults is multiplied with the number of steps of the simulation.

5.4 Repair and Compensation

Diagnosis capabilities of CatIO can be extended by performing compensating or repair actions once diagnoses have been computed. For repair CatIO makes use of a `Controller` that receives a set of possible diagnosis, and which performs repair or compensating actions during simulation. In case of a single diagnosis CatIO used a mapping of diagnoses to repair actions, whereas otherwise a designer has to add some selection method for obtaining a single working hypothesis. A working hypothesis is a diagnosis which is assumed to be the real cause of the fault. Actions are performed by sending respective input values to the simulation over a user-defined number of steps for doing repair or compensation.

To illustrate repair and compensation, we make use of the following simulation scenario, which is given in Table 1. The initial position of the differential drive robot is located at the $(0, 0)$ point of the coordinate system. In a fault free scenario robot has to move straight on the x axis. In the simulation scenario shown in Table 1 we inject a fault at time step 5. Up to this point, the robot has moved 15th feet along the x axis from the initial point. For the next 10 steps the fault is manifested by tilting the robot's heading angle to the left. At time step 15 of the simulation fault is corrected and robot is moving once again in a straight line but with compromised direction. Fault injected simulation without any repair or compensating actions is denoted by orange line in Figure 5.2.

When using the diagnostic model shown in Listing 2, we are able to compute $AbRightWheel$ as diagnosis after time step 5. Complete correction of heading angle is still not possible as it does not provide enough information about the fault. Therefore we only know that fault is in the right wheel, and we can consequently perform a repair action. The repair action is a single step action that restores the behavior of the component back to correct state. The grey line in Figure 5.2 depicts the movement of the robot when using the repair action. Note that in systems whose behavior or goal is not time-dependent repair actions would suffice.

In contrast to repair actions, a compensating action can completely restore the heading angle that is given in the blue line in the Figure 5.2. Such a compensating action requires an exact fault mode of the faulty component. The union of the models shown in Section 5.2 enables such a computation of a single diagnosis. For the scenario shown in Table 1 the diagnosis is $RightWheelFaster$. The compensating action of restoring the heading angle is performed in 4 time steps (first by making the heading angle parallel with the x axis and then performing steps which are inverse of the fault). Hence, compensating actions are usually requiring

a sequence of actions over time. In CatIO both repair and compensating actions have to be specified as programs that correspond to diagnosis candidates.

5.5 Further capabilities of CatIO

In this manuscript, we focused on consistency-based diagnosis with CatIO. However, abductive diagnosis based on ATMS[12] is also fully supported by CatIO. For more details about abductive diagnosis and the automatic generation of abductive model we refer interested readers to [10].The automatic generation of abductive models is fully supported by CatIO. The dynamic generation of simulation scenarios outlined in Section 5.1 alongside with the dedicated `Diff` interface, enables the designer to automatically generate rules of the form *FaultMode(C_1), ... FaultMode(C_n)→ effect*. In CatIO automatically generated models can be manually expanded by adding additional rules or by stating physical impossibilities.

CatIO offers also a graphical plotting of simulation values. Plotting can be performed on variable pairs or on a single variable that is plotted over time. The content of the CatIO graphical user interface that aids the designer is outlined in [8].

6 Conclusion and Future Work

CatIO supports designers in implementing model-based diagnosis for cyber-physical system. The CatIO framework allows the designer to develop consistency-based and abductive diagnosis models. It allows further the comparison and evaluation of the developed models via performing experiments utilizing simulation models. Powerful simulation capabilities of Modelica coupled with principles of model-based diagnosis allow designers to reason about the proper level of abstraction needed for diagnosis models, design and development of repair actions as well as automatic generation of abductive models.

CatIO executes simulations via interaction with an FMI that can be created from Modelica models. FMI is an open standard and other modeling languages like MATLAB Simulink and Dymola can also export their models to an FMI. Therefore, CatIO is not limited to the Modelica modeling language. Integrating concepts discussed in [8] and also shown in this paper provide designers of cyber-physical systems with a single and integrated development environment in which both simulation models and diagnostic models can be created. Further improvements like the automated generation of repair and compensating actions from models or the coupling of CatIO with 3D simulation would further increase the usefulness of CatIO for practical applications.

In future work, we consider expanding the modeling languages used for diagnosis as well as adding more system description formats like constraint representation with accompanying solver. Exploring today's computational power and distributed computing by interfacing CatIO to be able to run multiple simulations in parallel would provide great run-time improvement for automatic generation of the abductive model's knowledge base as well. Therefore, we also plan extending CatIO to handle such concurrent simulation and computations.

Acknowledgment

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

[1] A. Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT*, page 2008.

[2] T. Blochwitz, M. Otter, J. ÃĚkesson, M. Arnold, C. ClauÃ§, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. 09 2012.

[3] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[4] J. De Kleer and B. C. Williams. Diagnosis with behavioral modes. In *11th International Joint Conference on Artificial Intelligence - Volume 2*, page 1324âĂŞ1330, 1989.

[5] G. Friedrich, G. Gottlob, and W. Nejdl. Hypothesis classification, abductive diagnosis and therapy. In *Proceedings of the International Workshop on Expert Systems in Engineering*, Vienna, September 1990. Springer Verlag, Lecture Notes in Artificial Intelligence, Vo. 462.

[6] P. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach*. John Wiley & Sons, 12 2014.

[7] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94–96, 2009.

[8] E. MuÅąkardin, I. Pill, and F. Wotawa. Catio - a framework for model-based diagnosis of cyber-physical systems. 2020.

[9] I. Pill and T. Quaritsch. RC-Tree: A variant avoiding all the redundancy in Reiter's minimal hitting set algorithm. In *IEEE Int. Symp. Software Reliability Engineering Workshops (ISSREW)*, pages 78–84, 2015.

[10] I. Pill and F. Wotawa. On using an i/o model for creating an abductive diagnosis model via combinatorial exploration, fault injection, and simulation. In *29th International Workshop on Principles of Diagnosis (DX'18)*, 2018. http://ceur-ws.org/Vol-2289/paper9.pdf.

[11] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[12] R. Reiter and J. Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. pages 183–189, 01 1987.

[13] M. Sayed-Mouchaweh (editor). *Diagnosability, Security and Safety of Hybrid Dynamic and Cyber-Physical Systems*. Springer Publishing Company, Incorporated, 1st edition, 2018.

[14] J. M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs against Errors*. John Wiley & Sons, Inc., 1997.

[15] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375, March 2013.